Efficient DNN Inference on Edge Devices: Term-Paper Presentation

Rambha Satvik and Pandrangi Aditya Sriram

October 3, 2025

Outline

Power Efficient ASIC for ViTs

Naveen R and Sudhish N George

28th International Symposium on VLSI Design and Test (VLSIDT), 2024

Motivation

- Edge AI systems need to run complex DNNs under tight memory, latency, and power constraints.
- Vision Transformers (ViTs) offer strong accuracy but are resource-hungry.
- We review frameworks that optimize ViT inference on edge devices via hardware and system-level techniques.

ViT ASIC: Motivation

- In edge environments, there is typically a tight power constraint.
- Vision Transformers are gaining huge attention in the field of Computer Vision due to their accuracy.
- ViTs employ MHA - very compute-expensive and power-hungry.
- Majority of the time is spent on softmax computation.
- Proposed ASIC simplifies softmax overhead via Taylor approximation.

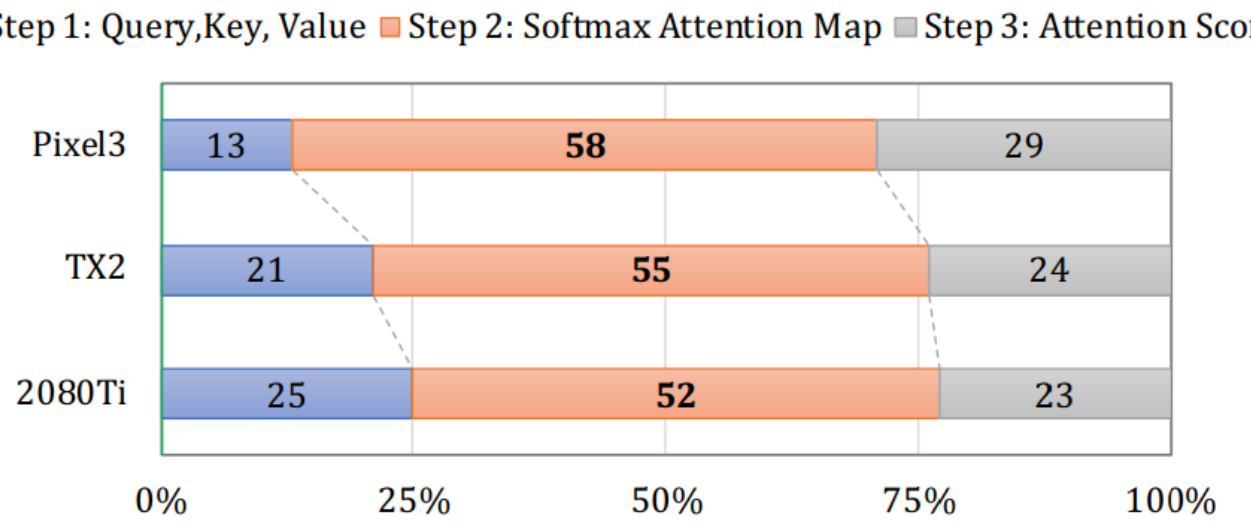


Figure: Runtime Simulation of MHA [**?**]

Background - ViT

- Patch embeddings (including a [CLS] token) are fed into a transformer encoder block
- Final MLP head for classification.
- Positional embeddings are added to the vectorized patches to tell the position of the patch to the model.
- The transformer consists of alternating attention layers and FFNNs
- Residual Connections

Background - ViT

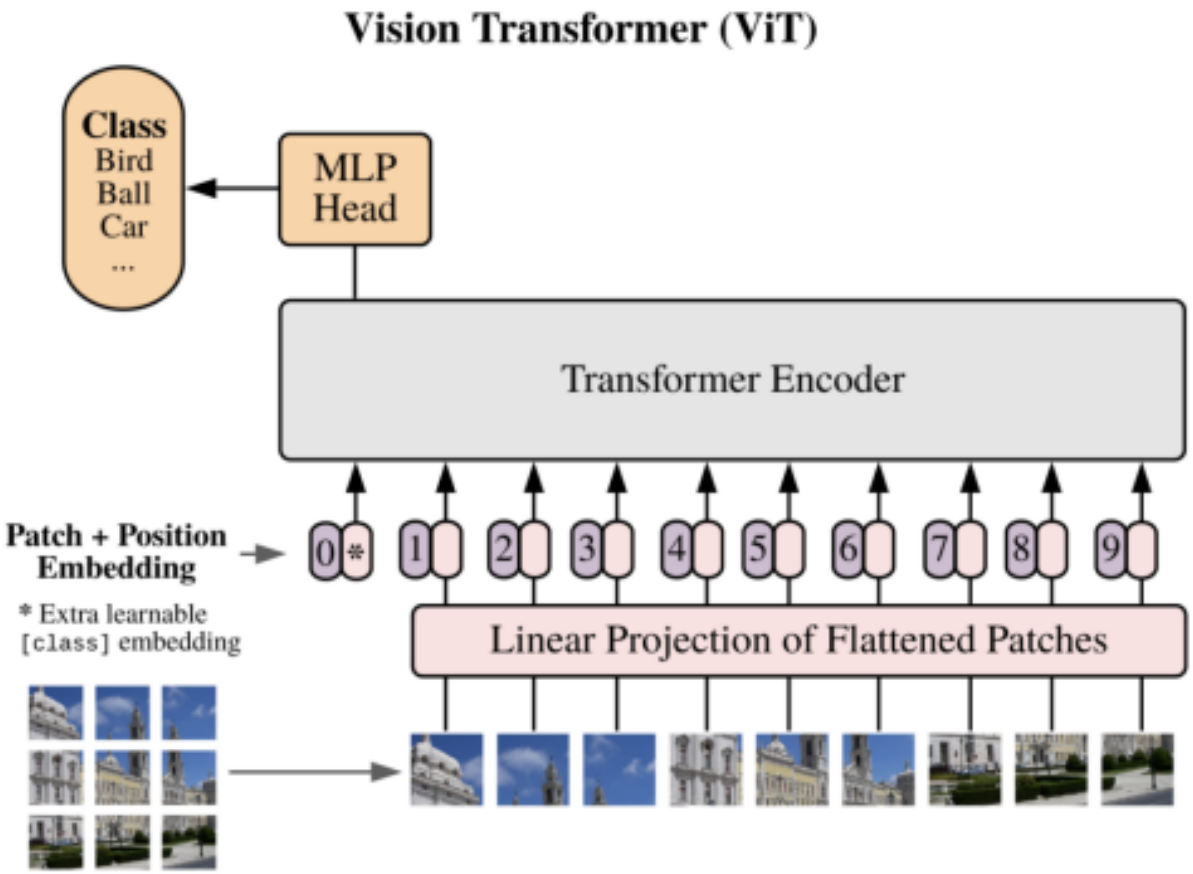


Figure: Vision Transformer Overview [?]

ASIC Innovations

- A **software optimization** replacing the costly softmax in attention with a Taylor expansion of the first order (as in ViTALiTy), reducing computational complexity
- A **hardware design** for self-attention using a Systolic Array Matrix Multiplier (SAMM) that accelerates the large matrix multiplications in multi-head attention.
- The use of low-power techniques, such as clock gating, in the ASIC implementation, yielding roughly 25.8% dynamic power reduction with modest area/frequency overhead.

Software Optimization

Vision Transformer (ViT) splits image $x \in \mathbb{R}^{H \times W \times C}$ into $P \times P$ patches $\Rightarrow x \in \mathbb{R}^{N \times P^2 \times C}$, where $N = \frac{HW}{P^2}$. Each layer uses multi-head attention:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Challenge: Softmax is compute-intensive [**?**]. Standard ViT has $O(n^2 d)$ complexity.

Software Optimization

Alternatives: ViTALiTy: Uses first-order Taylor approximation with mean-centered keys:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) \approx \text{diag}^{-1}\left(n\sqrt{d}\mathbf{1} + Q\hat{K}_{sum}^T\right)\left(\sqrt{d}\mathbf{1}\mathbf{1}^T + Q\hat{K}^T\right)$$

Result: ViTALiTy achieves 95.7% on MNIST with 37.7% faster inference and fewer FLOPs. Hardware is optimized for ViTALiTy.

Hardware Architecture of ASIC

- Pre-processing stage which involves patch embedding, positional encoding, and weight quantization to 16-bit fixed point.
- **Processing Units (PUs)** containing multiplier and an adder - 2D grid - pipelined systolic array. Provide high throughput and energy-efficient operation.
- **Column Accumulator:** Column-wise mean-centering of the key matrix before softmax.
- **Controller and Memory:** Control the data flow through various instructions - selection of weights, data reuse, reducing power consumption, seamless data transfer between memory.
- On-chip memory buffers store the embedded input patches and the weight matrices.

Hardware Architecture of ASIC

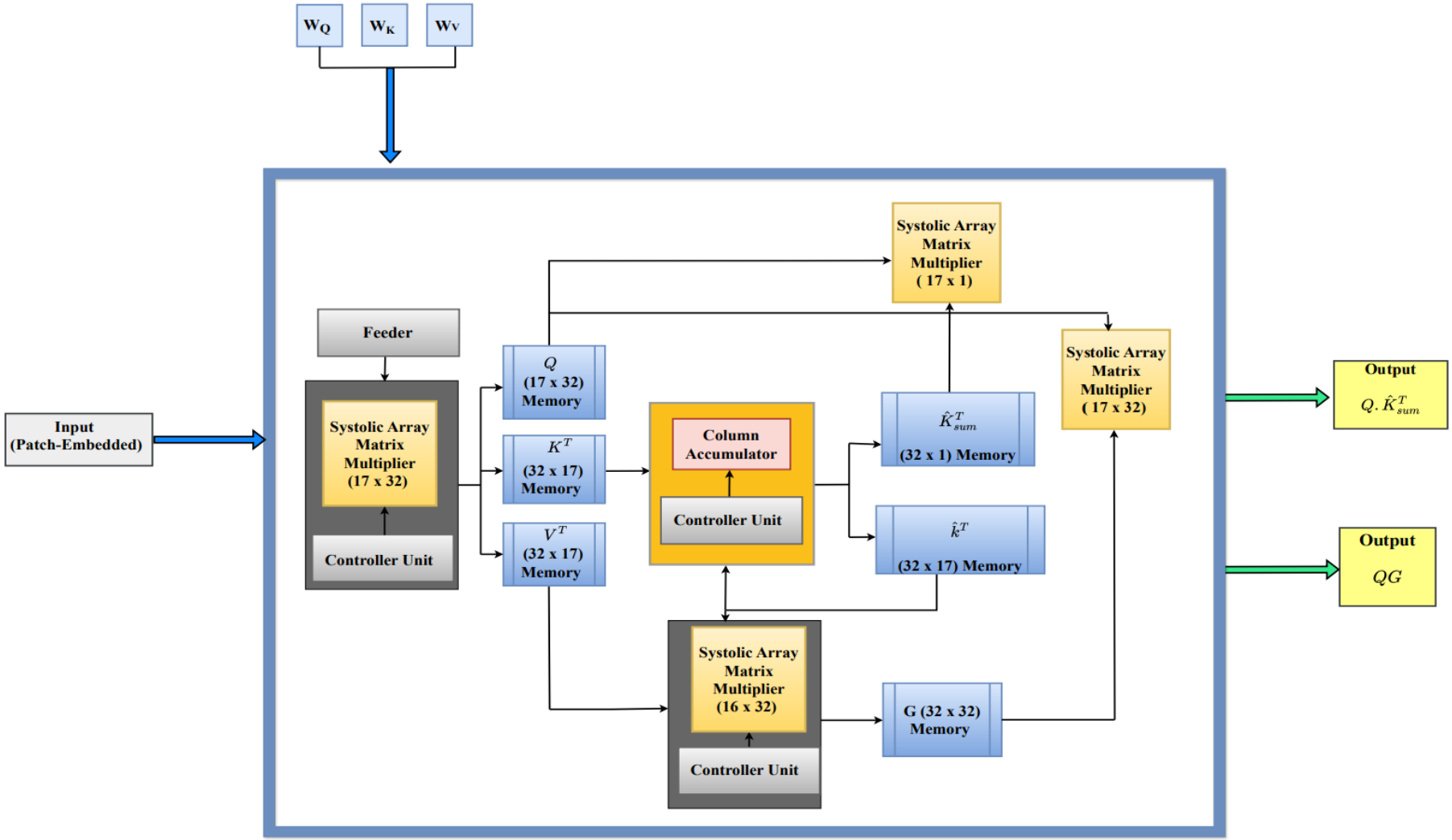


Figure: SAMM-Based Hardware Design [?]

ASIC: Evaluation

- **Power:** 25.8% lower dynamic power vs DianNao, 50% vs Eyeriss
- **Area/Frequency:** 180nm at 100MHz, scalable to 250MHz at 90nm.
- **Accuracy:** Matches 95% MNIST accuracy with lower power.

Reconfigurable Array Processor-based Accelerator

Rui Shan • Yifan Zheng

Proceedings of the 2023 AIPR Conference, pp. 395–401

Reconfigurable ViT: Motivation

- ViT MHA is compute- and memory-intensive.
- As the number of heads and patches increases, the volume of matrix multiplications and nonlinear Softmax operations grows rapidly. High throughput and on-chip memory bandwidth is demanded.
- Goal: Real-time execution on edge reconfigurable hardware.

Core Contributions

- Dynamically reconfigurable processor array.
- 4-way concurrent attention heads via Processing Element Groups.
- Algorithmic improvements to softmax computation.
- Instruction-driven reconfiguration pipelines alternating attention/FFNN layers.

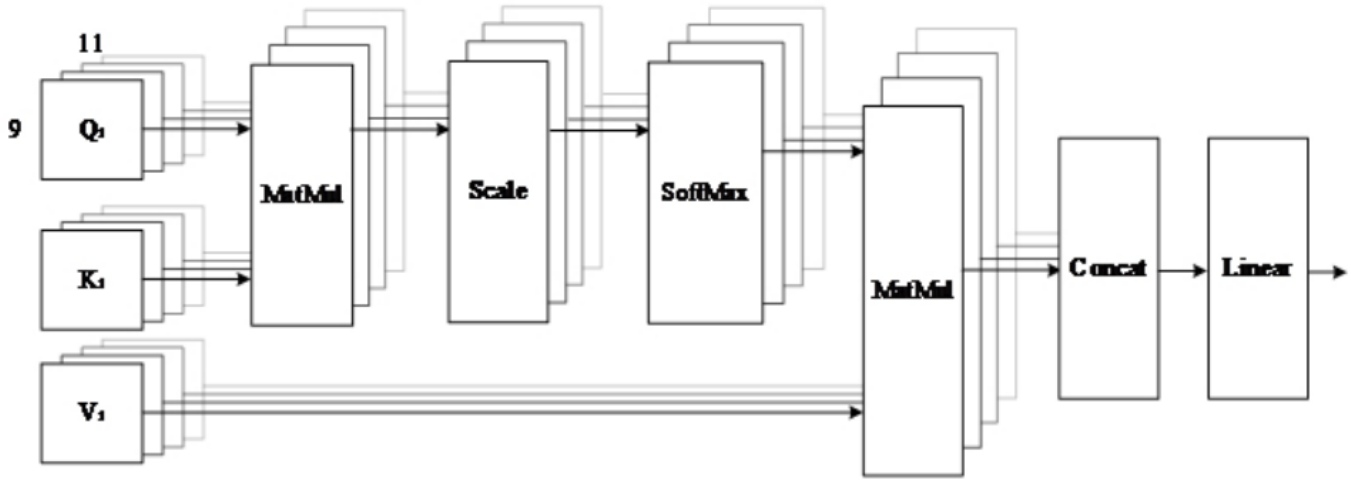


Figure: Multi-Head Attention Workflow [?]

Softmax Optimization

Hardware-friendly scheme that uses the log-sum-exp trick (subtract mean from exponent) to avoid direct exponent/division units.

$$Q = Q_1 \times W_{Q_1} + Q_2 \times W_{Q_2} + Q_3 \times W_{Q_3} + Q_4 \times W_{Q_4} \quad (1)$$

$$K = K_1 \times W_{K_1} + K_2 \times W_{K_2} + K_3 \times W_{K_3} + K_4 \times W_{K_4} \quad (2)$$

$$V = V_1 \times W_{V_1} + V_2 \times W_{V_2} + V_3 \times W_{V_3} + V_4 \times W_{V_4} \quad (3)$$

$$Attention\,(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (4)$$

$$softmax\,(S_i) = \frac{e^{s_i}}{\sum_{n=1}^{N} e^{s_n}} \quad (5)$$

$$Attention\left(Q_1K_1^T, V\right) = \frac{Q_1K_1^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (6)$$

$$Attention\left(Q_2K_2^T, V\right) = \frac{Q_2K_2^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (7)$$

$$Attention\left(Q_3K_3^T, V\right) = \frac{Q_3K_3^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (8)$$

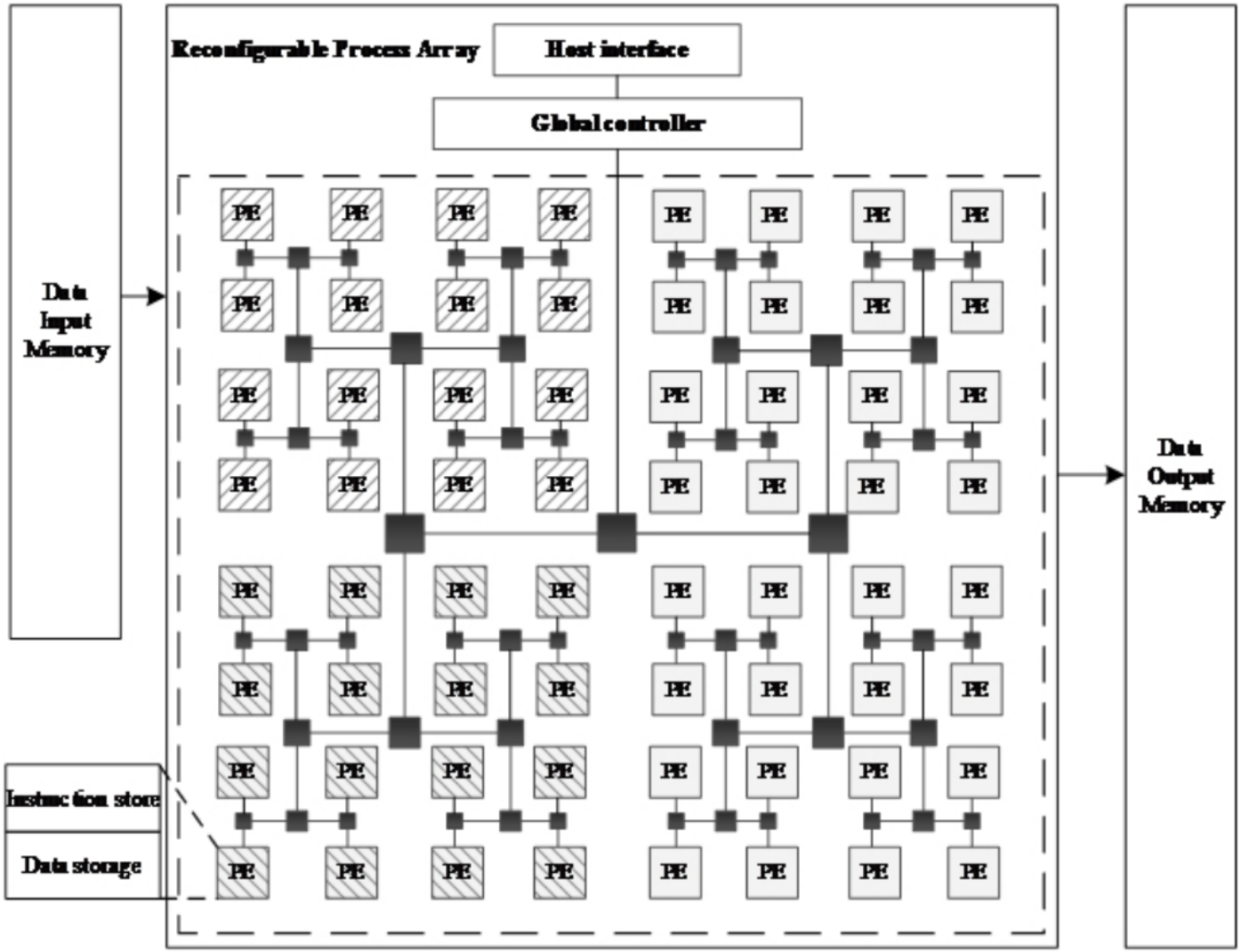
$$Attention\left(Q_4K_4^T, V\right) = \frac{Q_4K_4^T}{\sum_{i=1}^{I} Q_iK_i^T} \times V \quad (9)$$

Figure: Optimized Softmax and Attention [**?**]

Hardware Design Overview

- Host Interface (HI) and Global Controller (GC) that manage data and configuration
- 4×4 array of Processing Elements (PEs) organized into four 2×2 Processing Element Groups (PEGs). Each PEG can be configured independently.
- PEG00 cluster allocated to MHA, PEG10 for FFNN and residuals.
- Global Controller issues a sequence instructions (44 bits long) that dynamically repipelines the array.
- Handshake signals (sort of like interrupts) to switch clusters: once the 4-way self-attention in the first cluster completes, it triggers reconfiguration of the second cluster, and vice versa.
- Ensure no timing conflicts and all dependencies are satisfied.

Hardware Design Overview



Dataflow and Execution

- The input's channels are sequentially fed into the array, i.e. it performs attention on each channel in turns.
- Ensures that at most one frame's data is in the array at a time, avoiding buffer overflows.
- Aligns the data with the 4-way attention: each PEG computes one of the four heads on corresponding parts of the patch embeddings.
- This reorganization increases data locality and bandwidth.
- The dataflow graph shown in next slide maximizes MHA parallelism based on data flow maps and makes best use of hardware.
- No direct division is performed, instead bit shift operations and adders are utilized for both numerators and denominators.

Dataflow and Execution

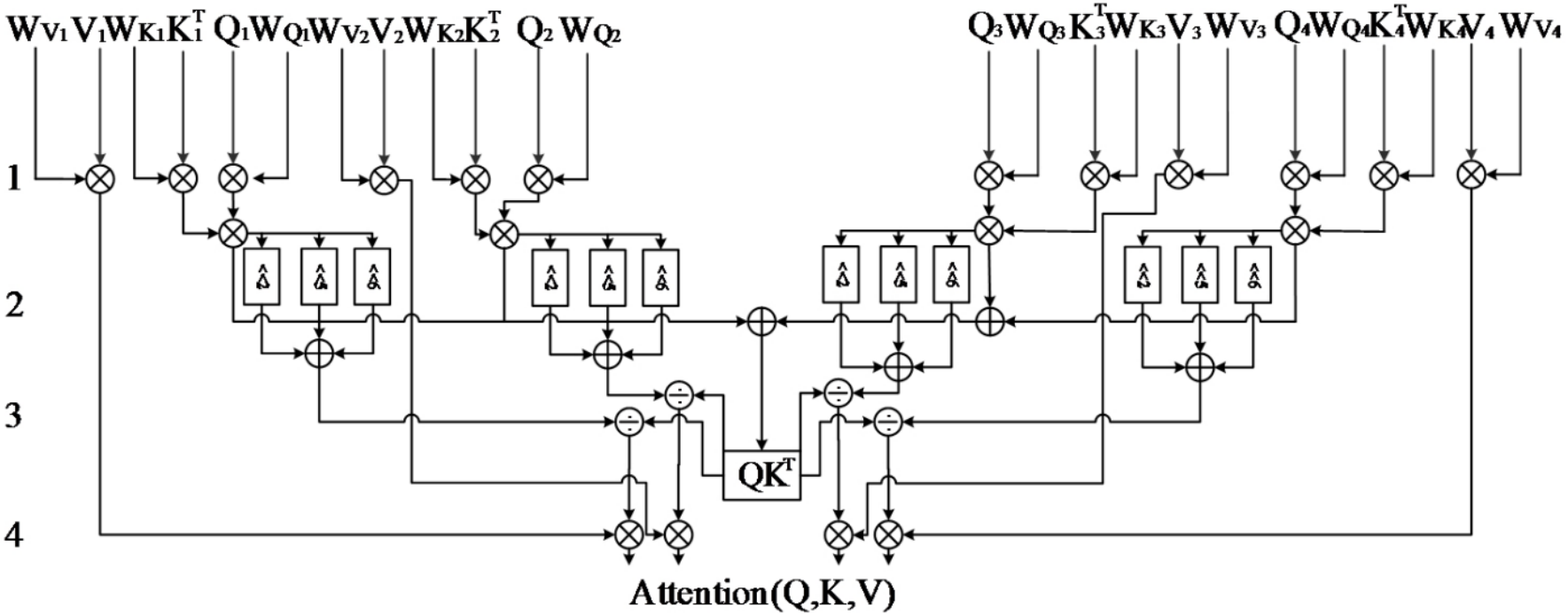


Figure: MHA Parallel Dataflow [**?**]

Reconfigurable Encoder Results

Hardware Resource	Ref.[10]	Ref.[11]	Ours
DSP	2,420	1,024 (512)	32
LUT	258,000	65,385 (132,433)	51,062
FF	257,000	31,739 (52,332)	32,757
BRAM	1,002	539 (439)	475
Platform	Alveo U50	ZCU102	XCVU440

Figure: Comparison with State-of-the-art

- **+0.61% accuracy**, **-7.29% latency**, **-17.09% resources**.

Conclusion

- Tailored hardware and runtime optimizations enable ViT inference on edge devices.
- Systolic arrays and log-sum-exp approximations improve both power and throughput.
- Both ASIC and reconfigurable designs show promising results in latency, accuracy, and utilization.

SwapNet

Kun Wang Jiani Cao Zimu Zhou Zhenjiang Li

IEEE Transactions on Mobile Computing (TMC), 2024

(Accepted January 2024)

SwapNet: Motivation (1/2)

- Edge AI devices (e.g., NVIDIA Jetson, Google Coral) have only 2–8 GB unified memory
  shared by OS, vision pipelines, and multiple DNNs.
- Real-world systems (autonomous vehicles, drones) often require running **multiple** DNNs concurrently (detection, segmentation, SLAM).

MEMORY ALLOCATION OF NON-DNN TASKS AND THE REMAINING MEMORY BUDGET FOR DNN TASKS ON AN EXAMPLE AUTONOMOUS VEHICLE APPLICATION

Tasks	Memory Usage	Percentage
Operating System	1038 MB	12.7%
SLAM and Navigation	1815 MB	22.2%
Map Repository	1229 MB	15.0%
Video Capture and Encoding	488 MB	5.9%
CUDA Kernel	1518 MB	18.5%
Remaining Memory	2104 MB	25.7%

SwapNet: Motivation (2/2)

- Traditional strategies:
  - Model compression: precision reduction, pruning *(loss vs. memory trade-off)*.
  - Cloud offloading: high latency, bandwidth, privacy issues.
  - Framework-level swapping (naive): heavy overhead, framework-internal copies.
- **Goal:** Enable on-device, *lossless* execution of models larger than physical memory.

SwapNet: Challenges (1/2)

- **Memory Bandwidth and Latency**: Frequent allocations/deallocations and memcpy cause stalls.
- **Redundant Data Movement**: Explicit cudaMemcpy calls even on unified-memory hardware.
- **Graph Continuity**: Partitioning DNN disrupts end-to-end execution flow.

SwapNet: Challenges (2/2)

- **Compatibility**: Minimal changes to existing PyTorch/CUDA code.
- **Multi-DNN Scheduling**: Run several oversized models with interleaved inference.
- **Accuracy and Latency**: Maintain near-native performance and precision.

SwapNet: Problems Addressed

- Memory budget overflow
  Can execute models whose total footprint exceeds device memory.
- Inefficient swapping
  Eliminates framework-induced copy and allocation overhead.
- Scheduler integration
  Interfaces with high-level schedulers for multi-model workloads.

SwapNet: Zero-Copy Swap-In

- Use CUDA Unified Memory APIs:
  - Map host-resident weight pages directly into GPU address space.
  - Avoid explicit cudaMemcpy latency.
- Achieves near-zero swap-in cost.

SwapNet: Model Assembly by Reference

- Maintain pointers to weight tensors:
  - Update module references when swapping blocks.
  - No deep-copy or re-allocation of weights.
- Transparent integration: No changes to user models.

SwapNet: Multi-DNN Scheduling Utility

- Provides block-level API: `load_block`, `run_block`, `unload_block`.
- Enables round-robin or priority-based schedulers for interleaved inference.
- Demonstrated on YOLOv5 and DeepLabV3, achieving $< 1.2\times$ combined latency for two models.
- Scales to multiple concurrent DNNs within memory limits.
- Scheduling of various blocks is done keeping latency and memory of the pertaining block in mind.
- Since, latency can't be calculated each time, a few linearly related parameters are chosen and linearly regressed.
- The latency of a layer is calculated using the obtained slopes and parameters.

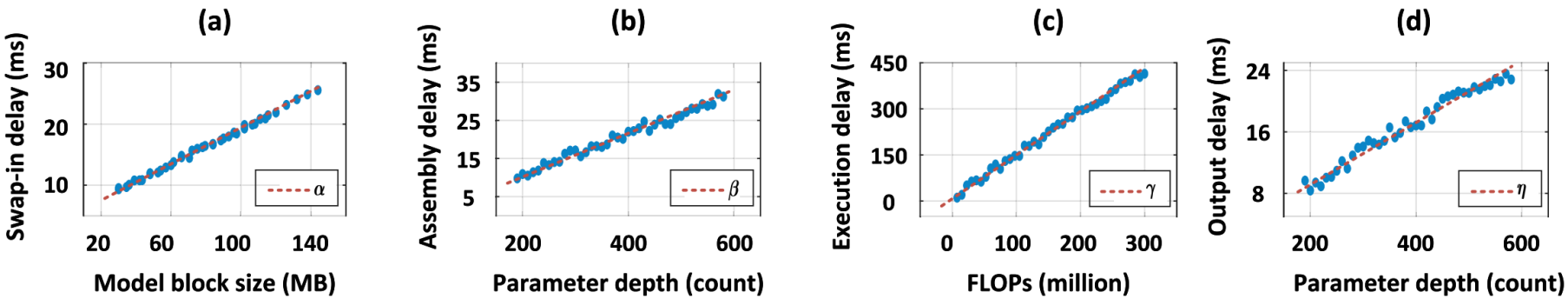


Figure: Regressing parameters

SwapNet: Key Contributions

1. **SwapNet middleware**: Transparent block swapping for oversized DNNs.
2. **Zero-copy and reference-based assembly**: Minimizes overhead.
3. **Evaluation on 11 tasks**: Handles up to 5.8× memory overcommit with 6.2% latency overhead and improved accuracy.
4. **Multi-model support**: Interleaved inference with near-native aggregate performance.

Hermes

Xueyuan Han Zinuo Cai Yichu Zhang
Chongxin Fan Junhan Liu Ruhui Ma Rajkumar Buyya

2024 IEEE 42nd International Conference on Computer Design (ICCD)

Hermes: Motivation

- Edge scenarios with unpredictable OS activity (e.g., garbage collection, driver interrupts).
- Memory fragmentation leads to **unusable** free regions despite total free memory.
- Need fine-grained control over memory placement for **predictable** low-latency inference.
- No pipeline scheme till date performed memory optimisations for inference.
- Desire adaptive strategies that react to runtime behavior of DNN workloads.

Hermes: Challenges

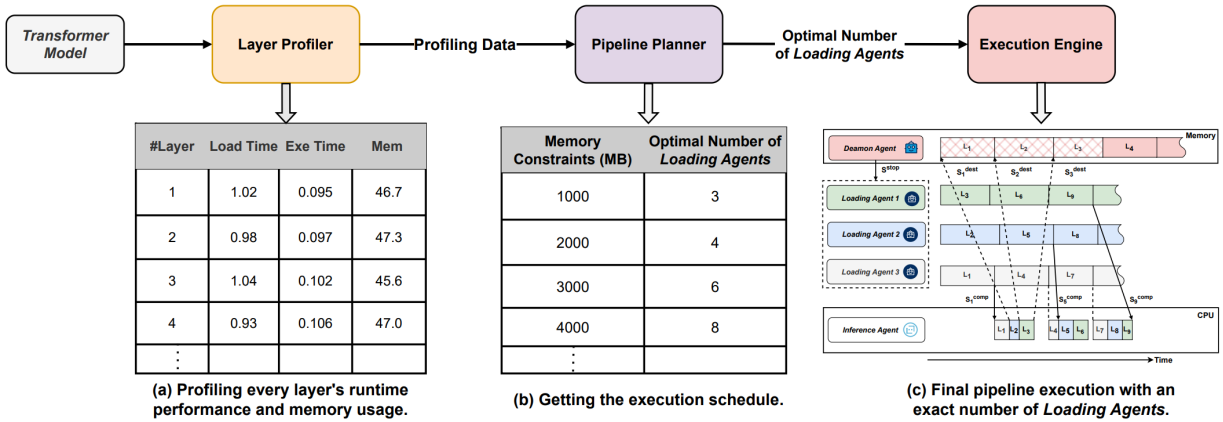
- **Dynamic Fragmentation**: Fragmentation arises over time from varied allocations.
- **Jitter and Jank**: OS/driver induced latency spikes (page faults).
- **Framework Hooks**: Must interpose on TensorFlow/PyTorch allocators with minimal user changes.

Hermes: Problems Addressed

- Unpredictable memory availability
  Provides contiguous blocks on demand.
- Excessive swap stalls
  Dynamic planning reduces unnecessary swaps.
- Integration complexity
  Seamless API to existing DNN frameworks.

Hermes: Layer Profiler and Pipeline Planner

- **Layer Profiler**: Monitors per-layer memory usage and execution time.
- **Pipeline Planner**: Constructs optimal execution order and swap schedule.
- Addresses unpredictable memory availability by planning around fragmentation.
- Balances the depth of the pipeline to minimize swap stalls.



(a) Profiling every layer's runtime performance and memory usage.

(b) Getting the execution schedule.

(c) Final pipeline execution with an exact number of *Loading Agents*.

Figure: Hermes system architecture

Hermes: Execution Engine

- Orchestrates memory allocation, swap-in/out, and compute tasks.
- Utilizes unified-memory accesses for low-latency swaps.
- Responds to runtime deviations to reduce swap-induced stalls.
- Solves excessive swap stalls through adaptive execution adjustments.

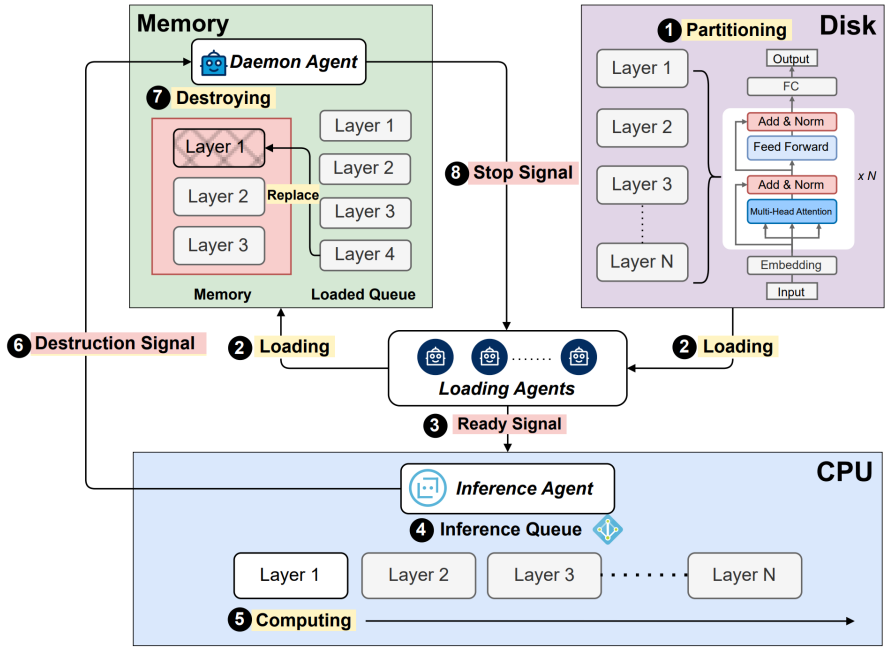


Figure: The overview and workflow of PIPELOAD

Hermes: Key Contributions

1. **Layer-level profiling and planning**: Predictable, fragmented-aware execution.
2. **Execution engine with adaptive adjustments**: Reduces swap stall latency by up to 50%.
3. **Runtime integration**: Minimal code changes for DNN frameworks.

THANK YOU!